10
Using X.509 Certificates in TLS

In *Chapter 8*, *X.509 Certificates and PKI*, we learned about **X.509 certificates**, while in *Chapter 9*, *Establishing TLS Connections and Sending Data over Them*, we learned about the **Transport Layer Security** (**TLS**) protocol and why certificates are important for TLS. We also reviewed some straightforward yet very popular usages of certificates in TLS, such as default server certificate verification performed by OpenSSL, hostname verification, and using an X.509 certificate to accept a TLS server connection.

In this chapter, we will expand on X.509 certificate usage in TLS and cover more advanced use cases.

In this chapter, we will cover the following topics:

- Custom verification of peer certificates in C programs
- Using Certificate Revocation Lists in C programs
- Using the Online Certificate Status Protocol
- Using TLS client certificates

Technical requirements

This chapter will contain commands that you can run on a command line and C source code that you can build and run. For the command line commands, you will need the `openssl` command-line tool, along with OpenSSL dynamic libraries. To build the C code, you will need OpenSSL dynamic or static libraries, library headers, a C compiler, and a linker.

We will implement some example programs in this chapter to practice the topics at hand. The full source code for those programs can be found at `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter10`.

Custom verification of peer certificates in C programs

Every TLS connection is established between two peers: the client and the server. Each peer can request and verify the other peer's certificate. In real life, the server certificate is almost always verified during the TLS handshake. Before TLS 1.3, the TLS protocol supported anonymous ciphers, which allowed the server to operate without a certificate. In practice, those anonymous ciphers were rarely used and were forbidden by default. So, in practice, a certificate for TLS has always been required. On the contrary, TLS client certificates are seldom used. However, verifying a client certificate in an application using OpenSSL is very similar to verifying a server certificate. Therefore, it makes sense to talk about peer certificate verification instead of limiting ourselves to server certificate verification. We will verify the server certificate in most of our code examples. More information about verifying client certificates will be provided later in the *Requesting and verifying a TLS client certificate on the server side programmatically* section.

You can verify the TLS peer certificate using OpenSSL in several ways:

- By using an OpenSSL built-in certificate verification code by calling `SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL)` or `SSL_set_verify(ctx, SSL_VERIFY_PEER, NULL)`, as we did in the `tls-client` program in *Chapter 9*, *Establishing TLS Connections and Sending Data over Them*. OpenSSL built-in verification can be adjusted by enabling optional checks such as hostname verification, by using the `SSL_set1_host()` function, or purpose verification, by using the `SSL_CTX_set_purpose()` or `SSL_set_purpose()` function.
- By combining the OpenSSL built-in verification code with your own verification callback function by calling `SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback)` or `SSL_set_verify(ctx, SSL_VERIFY_PEER, verify_callback)`. Note that the last function argument is not `NULL`, but a pointer to your verification callback function.
- By using a "big" certificate verification callback function instead of the OpenSSL built-in verification code by calling `SSL_CTX_set_cert_verify_callback(ctx, cert_verify_callback, arg)`.
- By getting and checking the peer certificate after the completed handshake, which you can do using functions such as `SSL_get_peer_certificate()`, `SSL_get_peer_cert_chain()`, and `SSL_get0_verified_chain()`. It is possible to force completion of the handshake by calling `SSL_CTX_set_verify()` or `SSL_set_verify()` with the `SSL_VERIFY_NONE` flag, or by always returning the successful return code from the verification callbacks.

In this section, we will write a small `verify-callback` program to demonstrate how to use the "small" verification callback, which can be set by the `SSL_CTX_set_verify()` or `SSL_set_verify()` function.

The following are some relevant `man` pages for functions that we are going to use:

```
$ man SSL_CTX_set_verify
$ man SSL_get_app_data
$ man X509_STORE_CTX_get_ex_data
$ man X509_STORE_CTX_get_error
$ man X509_STORE_CTX_get_current_cert
$ man X509_get_subject_name
$ man X509_NAME_print_ex
$ man BIO_s_mem
```

As we can read on the `SSL_CTX_set_verify()` man page, a verification callback function must have the following synopsis:

```
int SSL_verify_cb(
    int preverify_ok, X509_STORE_CTX* x509_store_ctx);
```

This function takes the following parameters:

- `preverify_ok`: Error indicator
- `x509_store_ctx`: X.509 certificate verification context

The verification callback will be called at least once for every certificate in the peer certificate chain. The first call will be made for the last certificate in the chain, which will be a root CA certificate if the certificate chain can be completed. Then, OpenSSL will work toward the beginning of the chain, calling the verification callback for all intermediate certificates and finally for the peer certificate.

The OpenSSL library has a concept known as **certificate depth**. The depth of a certificate is a number that denotes how far that certificate is from the peer certificate in the certificate verification chain. The peer certificate always has a depth of 0. Its issuer certificate has a depth of 1, the issuer of the issuer has a depth of 2, and so on. There is also a concept known as **verify depth**, which specifies the depth or length of the whole certificate verification chain. It is possible to limit the maximum verify depth using the `SSL_CTX_set_verify_depth()` and `SSL_set_verify_depth()` functions.

When the certificate verification callback is called, the `x509_store_ctx` parameter will contain information about the certificate from the verification chain that's currently being checked. It is possible to get the current certificate using the `X509_STORE_CTX_get_current_cert()` function, and its depth using the `X509_STORE_CTX_get_error_depth()` function.

The `preverify_ok` parameter of the verification callback tells us whether OpenSSL found an issue with the current certificate. The `preverify_ok` parameter can have either a value of `0` or `1`. If `preverify_ok` is `1`, then OpenSSL has found no issues with the current certificate and will advance to the next certificate. If `preverify_ok` has a value of `0`, this means that OpenSSL has

detected an issue with the current certificate. In such a case, it is possible to get the error code using the `X509_STORE_CTX_get_error()` function and the error string representation using the `X509_verify_cert_error_string()` function. Also, OpenSSL will add an error to the error queue. As mentioned previously, it's important to process and clear the error queue – do not forget to do it.

Note that OpenSSL can call the verification callback with `preverify_ok=0` several times for one certificate if the certificate has several errors. It is also nice to know that if the verification callback is called with `preverify_ok=1`, then the `X509_STORE_CTX_get_error()` function will not necessarily return `X509_V_OK`, which is an indication of success. If the verification callback was previously called with `preverify_ok=0`, then `X509_STORE_CTX_get_error()` will return the last known error. Only if there were no verification errors with the current and the previous certificates will the function return `X509_V_OK`.

The verification callback should return either `1`, indicating success, or `0`, indicating failure. If `0` is returned, the verification fails immediately, a TLS alert message with an error is sent to the peer, and the TLS handshake fails. If `1` is returned, the verification continues. If all the calls of the verification callback return `1`, the verification succeeds.

It's worth mentioning that if `NULL` is provided as the verification callback via the `SSL_CTX_set_verify()` or `SSL_set_verify()` function, then the default verification callback is used. The default verification callback just returns `preverify_ok`.

Note that the verification callback function does not receive a `void*` argument pointing to user data, as many other callback functions do. Thus, if we want to avoid using global variables – and avoiding them is a good practice – it is not trivial to use data from the rest of the application in the verification callback. However, it is possible. Fortunately, we can get user data via the `x509_store_ctx` parameter, which is the verification context. It is possible to get the `SSL*` pointer from `x509_store_ctx` and then get the user data from the `SSL` object. The following code example will demonstrate how to do this.

Now, let's write a small `verify-callback` program that will demonstrate the usage of a verification callback function. The `verify-callback` program will connect to a TLS server, initiate a TLS handshake, print information about the peer certificate chain verification with the help of the verification callback, and disconnect. We are going to base our program on the `tls-client` program from *Chapter 9*, *Establishing TLS Connections and Sending Data over Them*. We will take the `tls-client` program code and develop it further.

The `verify-callback` program will take the same three command-line arguments as the `tls-client` program:

1. Server hostname
2. Server port
3. Optional argument: The name of the file containing one or more trusted CA certificates for server certificate verification

Registering the verification callback

In the code of our new `verify-callback` program, we will introduce a function with the following signature:

```
int verify_callback(
    int preverify_ok, X509_STORE_CTX* x509_store_ctx);
```

This function will be our verification callback.

In the `tls-client` program, we had the following line of code, which enabled the peer certificate verification process:

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
```

We will replace that code line with the following line, which will set the verification callback in the `SSL_CTX` object:

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
```

We will also set some user data for the verification callback. We will not set the user data in the `SSL_CTX` object; instead, we will set it in the `SSL` object. By doing this, it will be easier to get the user data in the `verify_callback` function code afterward. We can set any pointer as user data. In this example, our user data will be the `FILE* error_stream` pointer, which is the file handle for printing errors and diagnostic information. The following code line shows how we can set the user data:

```
SSL_set_app_data(ssl, error_stream);
```

`SSL_set_app_data()` is a macro for calling the `SSL_set_ex_data()` function, which is a member of the `*_set_ex_data()` function family. By using `*_set_ex_data()` functions, it is possible to set one or several so-called **extra data** pointers to objects of the `SSL`, `BIO`, and `X509` types, among others. You can read more about those functions on the `SSL_set_app_data` and `CRYPTO_set_ex_data` man pages. I find the `*_ex_data` API inconvenient and confusing, so I recommend using it as little as possible. If you need to set several user data elements to an SSL object, I recommend creating a structure, putting those data elements into the structure, and setting a pointer to that structure to an `SSL` object using the `SSL_set_app_data()` function, as we have done in this example.

In this example, we are interested in certificate verification, so we will not exchange request and response data with the server. Instead, we will shut down the connection after a successful or unsuccessful handshake. Because of that, we will remove the HTTP request-response code from the code inherited from the `tls-client` program.

Implementing the verification callback

Now, let's implement the verification callback:

1. Our verification callback will just print diagnostic information on the call parameters and the current certificate. We will print that information to the `FILE*` stream that we have set as the user data for the `SSL` object.

   We will start by implementing the verification callback from the function headline:

   ```
   int verify_callback(
       int preverify_ok, X509_STORE_CTX* x509_store_ctx) {
   ```

2. The first thing that we will do is get the user data, which is our error and diagnostic stream:

   ```
   int ssl_ex_data_idx = SSL_get_ex_data_X509_STORE_CTX_
   idx();
   SSL* ssl = X509_STORE_CTX_get_ex_data(
       x509_store_ctx, ssl_ex_data_idx);
   FILE* error_stream = SSL_get_app_data(ssl);
   ```

   To get the user data, which in this case is the `FILE*` stream, we have to get the `SSL` object. The `SSL` object is available as extra data with a particular index in the `x509_store_ctx` parameter. Hence, first, we have to get the index, then get the `SSL*` pointer, then the `FILE* error_stream` user data pointer.

3. Next, we will get the current certificate verification depth and the current error with the certificate, if any:

   ```
   int depth = X509_STORE_CTX_get_error_depth(x509_store_
   ctx);
   int error_code = preverify_ok ?
       X509_V_OK :
       X509_STORE_CTX_get_error(x509_store_ctx);
   const char* error_string =
       X509_verify_cert_error_string(error_code);
   ```

4. Next, we'll get the current certificate and its Subject:

   ```
   X509* current_cert =
       X509_STORE_CTX_get_current_cert(x509_store_ctx);
   X509_NAME* current_cert_subject =
       X509_get_subject_name(current_cert);
   ```

   Here, the Subject is an `X509_NAME` object.

5. Now, let's get a text representation of the obtained `X509_NAME` object by printing it to a memory BIO:

```
BIO* mem_bio = BIO_new(BIO_s_mem());
X509_NAME_print_ex(
    mem_bio,
    current_cert_subject,
    0,
    XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB);
```

6. Next, we will get a pointer to the text printed to the memory BIO and the text length:

```
char* bio_data = NULL;
long bio_data_len = BIO_get_mem_data(mem_bio, &bio_data);
```

Note that the printed text is not null terminated and may contain null characters, which is why knowing the text length is important.

7. Now that we have enough data to print the diagnostic information that we want, let's print it to `error_stream`:

```
fprintf(
    error_stream,
    "verify_callback() called with depth=%i, "
    "preverify_ok=%i, error_code=%i, error_string=%s\n",
    depth,
    preverify_ok,
    error_code,
    error_string);
fprintf(error_stream, "Certificate Subject: ");
fwrite(bio_data, 1, bio_data_len, error_stream);
fprintf(error_stream, "\n");
```

We could have printed the `X509_NAME` object containing the certificate Subject directly to `error_stream` using the `X509_NAME_print_ex_fp()` function. We haven't done this here because I wanted to demonstrate how to use memory BIO and get the text representation of an `X509_NAME` object. In real-world programs, we often need to get text into memory and process it further than just print it to a `FILE*` stream.

8. Our `verify_callback` function is almost finished. Now, we have to free some of the memory structures that we used:

```
BIO_free(mem_bio);
X509_free(current_cert);
```

   Note that we have not freed the `X509_NAME` object containing the certificate subject. This is because the `X509_NAME` object is owned by the `X509` object containing the certificate; the `X509_NAME` object will be freed when the `X509` object is freed. Generally, it is hard to predict which OpenSSL objects you must free and which you must not. Usually, you can find that information in the OpenSSL documentation, but not always. If you cannot find it in the documentation, you have to find the answer in another way, such as by checking the OpenSSL source code, experimenting, or asking your local OpenSSL expert.

9. After freeing the used data structures, we have to return the exit code from our verification callback. The easiest choice is to just return `preverify_ok`:

```
return preverify_ok;
}
```

   With that, you've learned how to make and use a certificate verification callback.

The complete source code for our `verify-callback` program can be found in this book's GitHub repository in the `verify-callback.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/verify-callback.c`.

Running the program

Let's run our program and see how it works. We will run it against the `www.example.org` server that we used previously. Before running it, please make sure that your current installation of OpenSSL will be able to find the trusted CA certificates. Please refer to *Chapter 9*, *Establishing TLS Connections and Sending Data over Them*, for instructions on how to provide trusted CA certificates to OpenSSL.

Here is how we run the program:

```
$ ./verify-callback www.example.org 443
verify_callback() called with depth=2, preverify_ok=1, error_
code=0, error_string=ok
Certificate Subject: C = US, O = DigiCert Inc, OU = www.
digicert.com, CN = DigiCert Global Root CA
verify_callback() called with depth=1, preverify_ok=1, error_
code=0, error_string=ok
Certificate Subject: C = US, O = DigiCert Inc, CN = DigiCert
TLS RSA SHA256 2020 CA1
```

```
verify_callback() called with depth=0, preverify_ok=1, error_
code=0, error_string=ok
Certificate Subject: C = US, ST = California, L = Los Angeles,
O = Internet Corporation for Assigned Names and Numbers, CN =
www.example.org
TLS communication succeeded
```

As we can see, OpenSSL has built a certificate verification chain for the peer certificate and verified it, starting from the root CA certificate and ending with the server certificate. Here, our `verify-callback` program printed what we expected for a successful case. But what about a failing case?

For the failing case, we will run our `verify-callback` program against a server under the special `badssl.com` domain. This domain contains servers that have been configured to test TLS handshake failures. You can find a list of different test servers and failure reasons at `https://badssl.com/`. In the following example, we will use the `incomplete-chain.badssl.com` server. We are expecting that OpenSSL will not be able to build a verification chain for the peer certificate because it will not be able to find an issuer certificate for the certificate in the chain. Let's check whether our expectation is correct:

```
$ ./verify-callback incomplete-chain.badssl.com 443
verify_callback() called with depth=0, preverify_ok=0, error_
code=20, error_string=unable to get local issuer certificate
Certificate Subject: C = US, ST = California, L = Walnut Creek,
O = Lucas Garron Torres, CN = *.badssl.com
Could not connect to server incomplete-chain.badssl.com on port
443
Errors from the OpenSSL error queue:
C0C158FD5D7F0000:error:0A000086:SSL routines:tls_post_process_
server_certificate:certificate verify failed:ssl/statem/statem_
clnt.c:1882:
C0C158FD5D7F0000:error:0A000197:SSL routines:SSL_
shutdown:shutdown while in init:ssl/ssl_lib.c:2242:
TLS communication failed
```

As we can see, our expectation was correct. We have got the following data in the verification callback:

```
depth=0, preverify_ok=0, error_code=20,
error_string=unable to get local issuer certificate
```

We have an issue (`preverify_ok=0`) with the server certificate (`depth=0`). Here, the issue is that OpenSSL could not find the issuer certificate (`error_code=20, error_string=unable to get local issuer certificate`).

From this, we can conclude that our `verify-callback` program works in both successful and failing cases and can diagnose TLS handshake issues with the help of the certificate verification callback.

Errors in the certificate chain are not the only reason for possible TLS handshake failures. Sometimes, the certificate's private key becomes compromised, and the certificate is revoked because of that. The certificate revocation status can be checked via **Certificate Revocation Lists** (**CRLs**) or the **Online Certificate Status Protocol** (**OCSP**). In the next section, we will learn about CRLs.

Using Certificate Revocation Lists in C programs

A CRL is a data structure that lists revoked certificates. A certificate can be revoked for several reasons, such as private key compromise, private key loss, an error in the certificate, cessation of operations by the certificate owner, the certificate being superseded by another certificate, and so on.

CRLs can often be downloaded from the **Certificate Authority** (**CA**) web servers. Those downloadable CRLs are often represented in **Distinguished Encoding Rules** (**DER**) format. For instance, a CRL for the `www.example.org` site certificate can be downloaded from `http://crl3.digicert.com/DigiCertTLSRSASHA2562020CA1-4.crl`.

Once downloaded to a file, a CRL can be viewed with the `openssl` command-line tool using the `openssl crl` subcommand:

```
$ openssl crl \
    -in DigiCertTLSRSASHA2562020CA1-4.crl \
    -inform DER \
    -noout \
    -text \
    | less
```

We will get the following output:

```
Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = DigiCert Inc,
                CN = DigiCert TLS RSA SHA256 2020 CA1
        Last Update: May 22 07:00:10 2022 GMT
        Next Update: May 29 07:00:10 2022 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                … hex bytes …
```

```
            X509v3 CRL Number:
                238
            X509v3 Issuing Distribution Point: critical
                Full Name:
                  URI:http://crl3.digicert.com/
DigiCertTLSRSASHA2562020CA1-4.crl
Revoked Certificates:
    Serial Number: 048A6B01889FA7BEF34AC92DAAC36079
        Revocation Date: Jun 22 00:31:11 2021 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    Serial Number: 098AB8A98137F3432A18DE8C1B7F6D2F
        Revocation Date: Jul  3 05:58:20 2021 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    … more certificates …
Signature Algorithm: sha256WithRSAEncryption
    … hex bytes …
```

As we can see, a CRL contains the following information:

- CRL format version
- The algorithm used to sign the CRL
- The CRL issuer in **Distinguished Name** (**DN**) format
- Last update timestamp
- Next update timestamp
- Optional CRL extensions
- List of revoked certificates
- Signature

A CRL must be signed by the issuer's private key. The issuer of the CRL is the same entity as the issuer of the certificate, which can be checked for revocation by the CRL. Only the certificate issuer has the authority to revoke the certificate by putting in a CRL.

Revoked certificates are identified by the certificate serial numbers. A revoked certificate entry in a CRL may contain a revocation reason, but it is not mandatory.

When it comes to checking a certificate against CRL programmatically, two approaches can be used:

- A CRL can be supplied to an `X509_STORE` object using the `X509_STORE_add_crl()` function. This is a good method if you already have a CRL for the target certificate. It is a good idea to cache CRLs as they may be quite large – up to tens of megabytes in size for large CAs.
- A CRL for a particular certificate can be looked up by a callback supplied by the `X509_STORE_set_lookup_crls()` function. This is a nice method if the checked certificate is not known in advance, which is a common situation for server certificates in TLS connections. We will use this method in the following code example.

Now, let's write a small `crl-check` program that will demonstrate how to use a CRL lookup callback function.

A CRL lookup callback function must have the following synopsis:

```
STACK_OF(X509_CRL)* X509_STORE_CTX_lookup_crls_fn(
    const X509_STORE_CTX* x509_store_ctx,
    const X509_NAME* x509_name);
```

This function takes the following parameters:

- `x509_store_ctx`: The X.509 certificate verification context
- `x509_name`: The subject of the certificate, whose revocation status should be checked using CRL

The function returns a stack of `X509_CRL` objects, which represent CRLs. OpenSSL often uses stacks as lists. As we can see, the callback function can return several CRLs if needed.

We are going to base our `crl-check` program on the `verify-callback` program, as described in the *Custom verification of peer certificates in C programs* section. We will take the `verify-callback` program code and develop it further.

Our `crl-check` program will take the same three command-line arguments as the `verify-callback` program:

1. Server hostname
2. Server port
3. Optional argument: Name of the file containing one or more trusted CA certificates for server certificate verification

We are going to use some OpenSSL functions that we have not used before. Here are the relevant `man` pages, along with the documentation for them:

```
$ man SSL_CTX_get_cert_store
$ man X509_STORE_set_lookup_crls
$ man X509_STORE_set_flags
$ man X509_NAME_print_ex_fp
$ man X509_get_ext_d2i
$ man ASN1_STRING_get0_data
$ man OSSL_HTTP_get
$ man d2i_X509_CRL_bio
```

Let's start the `crl-check` program implementation. First, we will register the CRL lookup callback.

Registering the CRL lookup callback

Here is how we register the CRL lookup callback:

1. In the code of our new `crl-check` program, we will introduce a function with the following signature:

   ```
   STACK_OF(X509_CRL)* lookup_crls(
       const X509_STORE_CTX* x509_store_ctx,
       const X509_NAME* x509_name);
   ```

   This function will be our CRL lookup callback function.

2. To ensure that our callback function will be called, we need to set it to the `X509_STORE` object using the `X509_STORE_set_lookup_crls()` function and enable CRL checking by setting the `X509_V_FLAG_CRL_CHECK` flag to the same `X509_STORE` object. The `X509_STORE` object is the certificate store of the `SSL_CTX` object. The `X509_STORE` object can store trusted certificates, untrusted certificates, CRLs, verification parameters, and extra application-specific data. We can get a pointer to the `X509_STORE` object from the `SSL_CTX` object.

   To set the callback function and enable the CRL check, we need to add the following code to our `run_tls_client()` function:

   ```
   X509_STORE* x509_store = SSL_CTX_get_cert_store(ctx);
   X509_STORE_set_lookup_crls(x509_store, lookup_crls);
   X509_STORE_set_flags(x509_store, X509_V_FLAG_CRL_CHECK);
   ```

The next step is to implement the CRL lookup callback.

Implementing the CRL lookup callback

Let's implement the CRL lookup callback function, `lookup_crls()`:

1. As we can see from the function signature, the callback function gets an `X509_NAME` object with the certificate Subject as the second parameter. If we had some CRL cache that could help us get the needed CRL based on the certificate Subject, that parameter would help us. But in this example, we don't have one. Instead, we are going to get the current certificate from the `X509_STORE_CTX` object, which was supplied as the first callback function parameter, get the CRL distribution points from the certificate, and download the relevant CRL from one of the distribution points.

   First, let's get the error stream in the same way as we did in the `verify_callback()` function:

   ```
   int ssl_ex_data_idx = SSL_get_ex_data_X509_STORE_CTX_
   idx();
   SSL* ssl = X509_STORE_CTX_get_ex_data(
       x509_store_ctx, ssl_ex_data_idx);
   FILE* error_stream = SSL_get_app_data(ssl);
   ```

2. Now, let's get the current certificate that needs to be checked for revocation:

   ```
   X509* current_cert =
       X509_STORE_CTX_get_current_cert(x509_store_ctx);
   ```

3. Next, let's get and print some useful information about the certificate:

   ```
   int depth = X509_STORE_CTX_get_error_depth(x509_store_
   ctx);
   X509_NAME* current_cert_subject =
       X509_get_subject_name(current_cert);
   fprintf(
       error_stream,
       "lookup_crls() called with depth=%i\n",
       depth);
   fprintf(error_stream, "Looking up CRL for certificate:
   ");
   X509_NAME_print_ex_fp(
       error_stream,
       current_cert_subject,
       0,
       XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB);
   fprintf(error_stream, "\n");
   ```

Note that this time, we used the `X509_NAME_print_ex_fp()` function instead of printing the certificate Subject to a BIO, as we did in the `verify_callback()` function.

4. The next step is to get the CRL distribution points from the certificate. Not every certificate defines CRL distribution points. Fortunately, the certificate from `www.example.org` that we are going to test defines them, so we can download the CRL from them. Here is how we get the CRL distribution points:

```
CRL_DIST_POINTS* crl_dist_points =
    (CRL_DIST_POINTS*) X509_get_ext_d2i(
        current_cert,
        NID_crl_distribution_points,
        NULL,
        NULL);
```

`CRL_DIST_POINTS` is a typedef for `STACK_OF(DIST_POINT)`, where `DIST_POINT` is a structure that defines a CRL distribution point.

5. Next, we must go through the distribution points, try to download the CRL from one of them, and return the downloaded CRL:

```
int crl_dist_point_count =
    sk_DIST_POINT_num(crl_dist_points);
for (int i = 0; i < crl_dist_point_count; i++) {
    DIST_POINT* dist_point =
        sk_DIST_POINT_value(crl_dist_points, i);
    X509_CRL* crl =
        download_crl_from_dist_point(
            dist_point, error_stream);
    if (!crl)
        continue;
    STACK_OF(X509_CRL)* crls = sk_X509_CRL_new_null();
    sk_X509_CRL_push(crls, crl);
    return crls;
}
return NULL;
```

That concludes the `lookup_crls()` function. As we can see, the `lookup_crls()` function will return `NULL` if the CRL could not be downloaded.

You have probably noticed that we call the `download_crl_from_dist_point()` function to download the CRL. We have to implement that function as well.

Implementing the function for downloading a CRL from a distribution point

Let's go through the function code for downloading the CRL:

1. The `download_crl_from_dist_point()` function has the following signature:

```
X509_CRL* download_crl_from_dist_point(
    const DIST_POINT* dist_point, FILE* error_stream);
```

2. The distribution point structure can contain several strings, called general names. Let's get them:

```
const DIST_POINT_NAME* dist_point_name =
    dist_point->distpoint;
if (!dist_point_name || dist_point_name->type != 0)
    return NULL;
const GENERAL_NAMES* general_names =
    dist_point_name->name.fullname;
if (!general_names)
    return NULL;
```

3. Then, we have to loop through the general names:

```
int general_name_count = sk_GENERAL_NAME_num(general_
names);
for (int i = 0; i < general_name_count; i++) {
    const GENERAL_NAME* general_name =
        sk_GENERAL_NAME_value(general_names, i);
    ...
}
```

4. The following code represents the code inside the loop. First, we will look for a URL among general names:

```
int general_name_type = 0;
const ASN1_STRING* general_name_asn1_string =
    (const ASN1_STRING*) GENERAL_NAME_get0_value(
        general_name, &general_name_type);
if (general_name_type != GEN_URI)
    continue;
```

5. If we find a general name with a URL, we will get its text representation:

```
const char* url =
    (const char*) ASN1_STRING_get0_data(
        general_name_asn1_string);
```

6. In this example, we only support HTTP URLs, so we will skip non-HTTP URLs. This is fine since CRL distribution points are usually HTTP URLs:

```
const char* http_url_prefix = "http://";
size_t http_url_prefix_len = strlen(http_url_prefix);
if (strncmp(url, http_url_prefix, http_url_prefix_len))
    continue;
```

7. At this point, we have found an HTTP URL that points to a CRL. Let's try to download it:

```
fprintf(error_stream, "Found CRL URL: %s\n", url);
X509_CRL* crl = download_crl_from_http_url(url);
```

8. If the downloading attempt has failed, we can try another general name:

```
if (!crl) {
    fprintf(
        error_stream,
        "Failed to download CRL from %s\n", url);
    continue;
}
```

9. If we have successfully downloaded the CRL, we will return it:

```
fprintf(error_stream, "Downloaded CRL from %s\n", url);
return crl;
```

That concludes the code inside the `for` loop.

If the `for` loop completes, this means that we could not download the CRL from the current CRL distribution point. In such a case, we will return `NULL`:

```
return NULL;
```

If the `download_crl_from_dist_point()` function returns `NULL`, the calling `lookup_crls()` function will try the next distribution point.

The `download_crl_from_dist_point()` function calls the `download_crl_from_http_url()` function to download the CRL from the found URL. Let's look at its code.

Implementing the function for downloading a CRL from an HTTP URL

The `download_crl_from_http_url()` function is relatively short:

```
X509_CRL* download_crl_from_http_url(const char* url) {
    BIO* bio = OSSL_HTTP_get(
        url,
        NULL /* proxy */,
        NULL /* no_proxy */,
        NULL /* wbio */,
        NULL /* rbio */,
        NULL /* bio_update_fn */,
        NULL /* arg */,
        65536 /* buf_size */,
        NULL /* headers */,
        NULL /* expected_content_type */,
        1 /* expect_asn1 */,
        50 * 1024 * 1024 /* max resp len */,
        60 /* timeout */);
    X509_CRL* crl = d2i_X509_CRL_bio(bio, NULL);
    BIO_free(bio);
    return crl;
}
```

In the `download_crl_from_http_url()` function, we have used two functions that we have not used before: `OSSL_HTTP_get()` and `d2i_X509_CRL_bio()`. `OSSL_HTTP_get()` is one of the functions of the OpenSSL HTTP client, a new functionality that has been introduced in OpenSSL 3.0. `d2i_X509_CRL_bio()` is a function that reads a DER-encoded CRL encoded from a BIO and converts it into an `X509_CRL` object. OpenSSL has many similar conversion functions, such as `i2d_X509()`, `d2i_RSAPublicKey_fp()`, and many others. The names of those functions consist of the following components:

- `d2i` or `i2d`, which specifies the direction of the conversion. `i` means "internal," an internal C structure in memory. `d` means "DER." Hence, `d2i` means "DER to internal" and `i2d` means "internal to DER."
- Object type – for example, `X509_CRL`.
- An optional suffix that denotes the input or output channel for reading or writing the DER-encoded object. The `bio` suffix means BIO, while the `fp` suffix means file pointer. If there is no suffix, the DER-encoded object will be read from or written to memory.

As mentioned previously, not all certificates define CRL distribution points. Even if the distribution points are defined, a CRL download may fail. What happens in such a case? If the `X509_V_FLAG_CRL_CHECK` flag is set and OpenSSL could not find a CRL, the certificate verification will fail with an `X509_V_ERR_UNABLE_TO_GET_CRL` error. Often, such a failure is unwanted because a very small share of the issued certificates is revoked. If the revocation check has failed, it's a much higher probability that the certificate has not been revoked. How can we avoid certificate verification failure if OpenSSL could not get the CRL? No flag instructs OpenSSL to ignore the `X509_V_ERR_UNABLE_TO_GET_CRL` error if the `X509_V_FLAG_CRL_CHECK` flag is set. However, all verification errors are processed by the verification callback, so it is possible to instruct OpenSSL to ignore that error by returning `1` from the verification callback. To do it, we just have to add the following lines to the `verify_callback()` function:

```
if (error_code == X509_V_ERR_UNABLE_TO_GET_CRL)
    return 1;
```

Those were the last few lines that we have to add to the `crl-check` program to make it complete.

The complete source code for our `crl-check` program can be found in this book's GitHub repository in the `crl-check.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/crl-check.c`.

Running the program

Let's run our program and see how it works. We will run it against the `www.example.org` server, which we also used for the previous example:

```
$ ./crl-check example.org 443
* lookup_crls() called with depth=0
  Looking up CRL for certificate: C = US, ST = California, L =
Los Angeles, O = Internet Corporation for Assigned Names and
Numbers, CN = www.example.org
  Found CRL URL: http://crl3.digicert.com/
DigiCertTLSRSASHA2562020CA1-4.crl
  Downloaded CRL from http://crl3.digicert.com/
DigiCertTLSRSASHA2562020CA1-4.crl
* verify_callback() called with depth=2, preverify_ok=1, error_
code=0, error_string=ok
  Certificate Subject: C = US, O = DigiCert Inc, OU = www.
digicert.com, CN = DigiCert Global Root CA
* verify_callback() called with depth=1, preverify_ok=1, error_
code=0, error_string=ok
  Certificate Subject: C = US, O = DigiCert Inc, CN = DigiCert
```

```
TLS RSA SHA256 2020 CA1
* verify_callback() called with depth=0, preverify_ok=1, error_
code=0, error_string=ok
  Certificate Subject: C = US, ST = California, L = Los
Angeles, O = Internet Corporation for Assigned Names and
Numbers, CN = www.example.org
TLS communication succeeded
```

As we can see, our `crl-check` program successfully found and downloaded the CRL so that OpenSSL could check certificate revocation against the downloaded CRL.

Let's also check a negative case – that is, when a certificate is revoked. We can find a revoked certificate by going to the `revoked.badssl.com` server:

```
$ ./crl-check revoked.badssl.com 443
* lookup_crls() called with depth=0
  Looking up CRL for certificate: CN = revoked.badssl.com
  Found CRL URL: http://crl3.digicert.com/
RapidSSLTLSDVRSAMixedSHA2562020CA-1.crl
  Downloaded CRL from http://crl3.digicert.com/
RapidSSLTLSDVRSAMixedSHA2562020CA-1.crl
* verify_callback() called with depth=0, preverify_ok=0, error_
code=23, error_string=certificate revoked
  Certificate Subject: CN = revoked.badssl.com
Could not connect to server revoked.badssl.com on port 443
Errors from the OpenSSL error queue:
C081A6A4617F0000:error:0A000086:SSL routines:tls_post_process_
server_certificate:certificate verify failed:ssl/statem/statem_
clnt.c:1882:
C081A6A4617F0000:error:0A000197:SSL routines:SSL_
shutdown:shutdown while in init:ssl/ssl_lib.c:2242:
TLS communication failed
```

As we can see, the server certificate verification failed because the certificate was revoked. We can see this in the `error_code=23, error_string=certificate revoked` error message.

The tests we've conducted confirm that our `crl-check` program works as expected and can detect revoked certificates.

With that, we have learned how to check certificate revocation using CRL. But checking CRL is not the only method of checking certificate revocation. There is also another revocation checking method, known as OCSP. We are going to review it in the next section.

Using the Online Certificate Status Protocol

In this section, we will learn about OCSP. First, we will learn what it is and how it works. Then, we will learn how to use OCSP on the command line and in C programs.

Understanding the Online Certificate Status Protocol

OCSP is a more modern method of certificate revocation checking that uses much less network traffic than CRL. When using OCSP, you don't need to download large CRL files. Instead, it is possible to query an OCSP server, also known as an **OCSP responder**, about the status of a particular certificate. Similar to how CRLs are published by the issuer of a particular certificate, OCSP servers are also maintained by the certificate issuer.

When querying an OCSP responder, an OCSP client sends an ASN.1-encoded OCSP request, containing a list of certificates to check for revocation. The OCSP server responds with an ASN.1-encoded OCSP response, which contains the queried certificate statuses, the validity period of the response, some other information, and the response signature that was produced using the certificate issuer's private key.

As OCSP responses are signed, OCSP responders usually operate via unencrypted HTTP. Choosing HTTP instead of HTTPS for OCSP checks also avoids circular OCSP requests when you're verifying the HTTPS server certificate of the OCSP responder.

OCSP is sometimes criticized for privacy issues. If a TLS client, such as a web browser, checks every TLS server certificate via OCSP, then OCSP responders know which websites the browser user visited. Each OCSP responder only knows about those websites that use certificates issued by the same issuer that owns the OCSP responder. However, if someone eavesdrops on the TLS client's internet line, they can read all the OCSP requests and responses that have been sent via unencrypted HTTP. On the other hand, the eavesdropper already knows the IP addresses of the servers that the user contacts. Hence, the eavesdropper is already in the position of knowing popular websites, such as Google or Facebook, just by knowing the server IP addresses. The additional information that the eavesdropper gains includes small websites hosted on shared web hostings, where the same IP can be used by several websites, and some blogging platforms where each blog has a subdomain.

Regardless of whether privacy issues are big or small, they can be mitigated via **OCSP stapling**. OCSP stapling means that the OCSP response about the server certificate is sent to the TLS client by the TLS server during the TLS handshake so that the TLS client does not need to contact the OCSP responder. In such cases, the OCSP response is stapled to the TLS handshake. Hence, this mechanism is called OCSP stapling. OCSP stapling is implemented using the **Certificate Status Request** TLS extension. The TLS client has to request the certificate status during the handshake to get it from the TLS server. The server, on the other hand, has to periodically poll the OCSP responder for the status of its certificate, to always have a fresh OCSP response and supply it to the clients.

OpenSSL supports OCSP querying both via the command line and programmatically. First, let's learn how to check certificates for revocation on the command line.

Using OCSP on the command line

OCSP querying on the command line can be performed using the `openssl ocsp` subcommand of the `openssl` tool. Let's try this functionality:

1. First, we will need the server certificate and its issuer certificate. We can get them using the `openssl s_client` subcommand:

```
$ echo | openssl s_client \
    -connect www.example.org:443 \
    -showcerts
```

   Note the `-showcerts` switch. This instructs the `openssl` tool to print certificates sent by the server to the terminal. The command will produce a long output that contains the certificates:

```
Certificate chain
 0 s:C = US, ST = California, L = Los Angeles, O =
Internet\C2\A0Corporation\C2\A0for\C2\A0Assigned\C2\
A0Names\C2\A0and\C2\A0Numbers, CN = www.example.org
   i:C = US, O = DigiCert Inc, CN = DigiCert TLS RSA
SHA256 2020 CA1
-----BEGIN CERTIFICATE-----
… base64 characters …
-----END CERTIFICATE-----
 1 s:C = US, O = DigiCert Inc, CN = DigiCert TLS RSA
SHA256 2020 CA1
   i:C = US, O = DigiCert Inc, OU = www.digicert.com, CN
= DigiCert Global Root CA
-----BEGIN CERTIFICATE-----
… base64 characters …
-----END CERTIFICATE-----
```

2. Now, we can copy and paste the first PEM-encoded certificates from the terminal to some files. Let's save the first printed certificate, with `CN = www.example.org`, to a file named `www.example.org.cert.pem` and the second certificate, with `CN = DigiCert TLS RSA SHA256 2020 CA1`, to a file named `DigiCert_Intermediate_CA1.pem`.

3. We have got the certificates – but how are we going to find out the OCSP responder URL? Fortunately, certificates often contain information about the OCSP responder URL in the **Authority Information Access** X509v3 extension. The certificate of the `www.example.org` website also contains that extension and the OCSP responder URL. We can extract this URL using the `-ocsp_uri` switch of the `openssl x509` subcommand:

```
$ openssl x509 \
    -in www.example.org.cert.pem \
```

```
    -noout \
    -ocsp_uri
http://ocsp.digicert.com
```

4. Then, we can perform a revocation check via OCSP using the following command:

```
$ openssl ocsp \
    -issuer DigiCert_Intermediate_CA1.pem \
    -cert www.example.org.cert.pem \
    -url http://ocsp.digicert.com
WARNING: no nonce in response
Response verify OK
www.example.org.cert.pem: good
        This Update: Jun 12 05:09:01 2022 GMT
        Next Update: Jun 19 04:24:01 2022 GMT
```

As we can see, the certificate is still `good`, which means it's not been revoked. The `Response verify OK` message means that the OCSP response has been verified successfully: it has been signed correctly by the expected issuer and is not outdated.

5. If we add the `-text` and `-resp_text` switches to the command line, we will be able to see the text representation of the OCSP request and response. Run the following command:

```
$ openssl ocsp \
    -issuer DigiCert_Intermediate_CA1.pem \
    -cert www.example.org.cert.pem \
    -url http://ocsp.digicert.com \
    -text \
    -resp_text
```

Running the preceding command will give you the following output. The first part of the output shows the OCSP request:

```
OCSP Request Data:
    Version: 1 (0x0)
    Requestor List:
        Certificate ID:
          Hash Algorithm: sha1
          Issuer Name Hash:
            E4E395A229D3D4C1C31FF0980C0B4EC0098AABD8
          Issuer Key Hash:
            B76BA2EAA8AA848C79EAB4DA0F98B2C59576B9F4
          Serial Number:
```

```
            0FAA63109307BC3D414892640CCD4D9A
    Request Extensions:
        OCSP Nonce:
            0410C5D14D224CCCEA68EEEC1478533D0DF8
```

The second part of the output shows the OCSP response and verification status of the response and the certificate:

```
OCSP Response Data:
    OCSP Response Status: successful (0x0)
    Response Type: Basic OCSP Response
    Version: 1 (0x0)
    Responder Id: B76BA2EAA8AA848C79EAB4DA0F98B2C59576B9F4
    Produced At: Jun 12 05:25:33 2022 GMT
    Responses:
    Certificate ID:
      Hash Algorithm: sha1
      Issuer Name Hash:
        E4E395A229D3D4C1C31FF0980C0B4EC0098AABD8
      Issuer Key Hash:
        B76BA2EAA8AA848C79EAB4DA0F98B2C59576B9F4
      Serial Number:
        0FAA63109307BC3D414892640CCD4D9A
    Cert Status: good
    This Update: Jun 12 05:09:01 2022 GMT
    Next Update: Jun 19 04:24:01 2022 GMT
    Signature Algorithm: sha256WithRSAEncryption
         … hex bytes …
WARNING: no nonce in response
Response verify OK
www.example.org.cert.pem: good
        This Update: Jun 12 05:09:01 2022 GMT
        Next Update: Jun 19 04:24:01 2022 GMT
```

As we can see, both OCSP requests and OCSP responses are rather simple. Both contain a compound certificate ID and an optional nonce. The response contains important additional data, such as certificate status, validity timestamps, and the signature.

Now that we've checked the revocation status of a good certificate via OCSP, let's take a look at a revoked certificate:

1. We already know that we can find a revoked certificate at `revoked.badssl.com`. As in the previous example, let's make a connection to the server and get the server certificate chain without the root CA certificate:

```
$ echo | openssl s_client \
    -connect revoked.badssl.com:443 \
    -showcerts
```

   Let's save the certificates from the command output to the terminal, the server certificate to the `revoked.badssl.com.cert.pem` file, and its issuer certificate to the `RapidSSL_Intermediate_CA1.pem` file.

2. Next, let's get the OCSP responder URL:

```
$ openssl x509 \
    -in revoked.badssl.com.cert.pem \
    -noout \
    -ocsp_uri
http://ocsp.digicert.com
```

3. Finally, let's check the server certificate revocation status:

```
$ openssl ocsp \
    -issuer RapidSSL_Intermediate_CA1.pem \
    -cert revoked.badssl.com.cert.pem \
    -url http://ocsp.digicert.com
WARNING: no nonce in response
Response verify OK
revoked.badssl.com.cert.pem: revoked
        This Update: Jun 12 01:09:02 2022 GMT
        Next Update: Jun 19 00:24:02 2022 GMT
        Revocation Time: Oct 27 21:38:48 2021 GMT
```

As we can see, this time, the certificate status is reported as `revoked`. Our observations confirm that the `openssl ocsp` subcommand can get the revocation statuses of both good and revoked certificates.

With that, we have learned how to use OCSP on the command line. Now, let's learn how to use OCSP programmatically.

Using OCSP in C programs

We can check the certificate revocation status programmatically in the following ways:

- Sending explicit OCSP requests to the OCSP responder. This method depends on the possibility to obtain the OCSP responder URL, such as via the **Authority Information Access** X509v3 extension in the server certificate.
- Using OCSP stapling and an OCSP stapling callback. This method depends on whether the TLS server has OCSP stapling support. We are going to use this method in the following example program.

Now, let's write a small `ocsp-check` program that demonstrates how to use an OCSP stapling callback function.

An OCSP stapling callback function must have the following synopsis:

```
int ocsp_callback(SSL* ssl, void* arg);
```

This function takes the following parameters:

- `ssl`: TLS connection object
- `arg`: User data pointer

The callback function should return one of the following values:

- A positive value if the stapled OCSP response is good
- `0` if the stapled OCSP response is bad
- A negative value if an error has occurred in the callback

To use the OCSP stapling callback, the Certificate Status Request TLS extension must be activated using the `SSL_CTX_set_tlsext_status_type()` function. The callback is set to the `SSL_CTX` object using the `SSL_CTX_set_tlsext_status_cb()` function. The user data pointer that will be passed to the callback function can be set using the `SSL_CTX_set_tlsext_status_arg()` function.

We are going to base our `ocsp-check` program on the `verify-callback` program, as described in the *Custom verification of peer certificates in C programs* section. We will take the `verify-callback` program code and develop it further.

Our `ocsp-check` program will take the same three command-line arguments as the `verify-callback` program:

1. Server hostname
2. Server port
3. Optional argument: Name of the file containing one or more trusted CA certificates for server certificate verification

We are going to use some OpenSSL functions that we have not used before. Here are the relevant `man` pages, along with the documentation for them:

```
$ man SSL_CTX_set_tlsext_status_type
$ man OCSP_response_status
$ man SSL_get0_verified_chain
$ man SSL_get_SSL_CTX
$ man SSL_CTX_get_cert_store
$ man OCSP_basic_verify
$ man OCSP_cert_to_id
```

Let's begin by implementing the `ocsp-check` program with the OCSP callback registration.

Registering the OCSP callback

Here is how we register the OCSP callback:

1. First, we will declare the following function. This will be our OCSP stapling callback:

   ```
   int ocsp_callback(SSL* ssl, void* arg);
   ```

2. In the `run_tls_client()` function, we have to activate the Certificate Status Request TLS extension and supply the OCSP stapling callback:

   ```
   SSL_CTX_set_tlsext_status_type(ctx, TLSEXT_STATUSTYPE_
   ocsp);
   SSL_CTX_set_tlsext_status_cb(ctx, ocsp_callback);
   ```

   Here, we could also supply a user data void pointer using the `SSL_CTX_set_tlsext_status_arg()` function, but we don't need it in this example. We already supply user data using the `SSL_set_app_data()` function.

   Those were the only code changes that we needed to supply and activate the OCSP stapling callback. Now, let's implement the callback itself.

Implementing the OCSP callback

Here is how we implement the OCSP callback:

1. We will start the callback's implementation by defining the default exit code for the function:

   ```
   int ocsp_callback(SSL* ssl, void* arg) {
       int exit_code = 1;
   ```

In this example, I want to give the maximum benefit of the doubt to the server certificate. This means that we will only consider the certificate to be revoked if it can be proven by the OCSP response. If something is wrong with the OCSP response – for example, it is unparseable or outdated – we will ignore the OCSP response and consider the server certificate as still valid. We're doing this because it is much more probable that some error occurred when the TLS server fetched the OCSP response from the OCSP responder than that the certificate has been revoked.

2. The first parameter that we receive in the OCSP stapling callback is the pointer to the `SSL` object, which stores information about the current TLS connection. This means that we don't have to get the `SSL` object in a sophisticated way as in the `verify_callback()` function. The SSL object is immediately available to us, and we can easily get our user data from it:

```
FILE* error_stream = SSL_get_app_data(ssl);
```

3. The next step is quite important – that is, getting the OCSP response:

```
const unsigned char* resp = NULL;
long resp_len = SSL_get_tlsext_status_ocsp_resp(ssl,
&resp);
if (resp_len <= 0 || !resp) {
    if (error_stream)
        fprintf(
            error_stream,
            "* ocsp_callback() called "
            "without OCSP response\n");
    goto cleanup;
}
```

The preceding code will get the OCSP response as DER-encoded data.

4. The next step is to decode that data into an `OCSP_RESPONSE` object:

```
OCSP_RESPONSE* ocsp_response =
    d2i_OCSP_RESPONSE(NULL, &resp, resp_len);
if (!ocsp_response) {
    if (error_stream)
        fprintf(
            error_stream,
            "* ocsp_callback() could not decode "
            "OCSP response\n");
    goto cleanup;
}
```

5. The next thing we can do is print the decoded OCSP response. As of OpenSSL 3.0, the OpenSSL library does not have a function that can print an OCSP response to a `FILE` stream. There is only the `OCSP_RESPONSE_print()` function, which prints the response to a BIO. Therefore, we will have to use the `FILE` pointer BIO:

```
if (error_stream) {
    BIO* bio = BIO_new_fp(error_stream, BIO_NOCLOSE);
    assert(bio);
    fprintf(
        error_stream,
        "* ocsp_callback() called "
        "with the following OCSP response:\n");
    fprintf(error_stream, "  -----\n  ");
    OCSP_RESPONSE_print(bio, ocsp_response, 0);
    fprintf(error_stream, "  -----\n");
    BIO_free(bio);
}
```

Note that we used the `BIO_NOCLOSE` flag when creating the BIO. This was quite important. Because of that flag, the later call to `BIO_free()` will not attempt to close the underlying `FILE` stream when we free the BIO.

6. The next step is to check the general status of the whole OCSP response:

```
int res = OCSP_response_status(ocsp_response);
if (res != OCSP_RESPONSE_STATUS_SUCCESSFUL) {
    if (error_stream)
        fprintf(
            error_stream,
            "OCSP response status is not successful\n");
     goto cleanup;
}
```

The OCSP response status tells us whether the OCSP responder could successfully process the OCSP request and form the OCSP response.

7. The next step is to verify the OCSP response signature:

```
OCSP_BASICRESP* ocsp_basicresp =
    OCSP_response_get1_basic(ocsp_response);
STACK_OF(X509)* verified_chain =
    SSL_get0_verified_chain(ssl);
```

```
SSL_CTX* ctx = SSL_get_SSL_CTX(ssl);
X509_STORE* x509_store = SSL_CTX_get_cert_store(ctx);
res = OCSP_basic_verify(
    ocsp_basicresp, verified_chain, x509_store, 0);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "OCSP response verification failed\n");
    goto cleanup;
}
```

Note the `SSL_get0_verified_chain()` function call. This function should only be called after the verification chain of the server certificate has been built; otherwise, the function may return a partial or incorrect chain. Fortunately, the OCSP callback is called after the server certificate has been verified, so we can safely call `SSL_get0_verified_chain()`.

8. The next thing that we have to do is find the status of the TLS server certificate in the OCSP response. An OCSP response can contain revocation information about several certificates, so we have to search for the status of the necessary certificate in the response:

```
X509* server_cert = sk_X509_value(verified_chain, 0);
X509* issuer_cert = sk_X509_value(verified_chain, 1);
OCSP_CERTID* server_cert_id =
    OCSP_cert_to_id(NULL, server_cert, issuer_cert);
ASN1_GENERALIZEDTIME* revocation_time  = NULL;
ASN1_GENERALIZEDTIME* this_update_time = NULL;
ASN1_GENERALIZEDTIME* next_update_time = NULL;
int revocation_status = V_OCSP_CERTSTATUS_UNKNOWN;
int revocation_reason = OCSP_REVOKED_STATUS_NOSTATUS;
res = OCSP_resp_find_status(
    ocsp_basicresp,
    server_cert_id,
    &revocation_status,
    &revocation_reason,
    &revocation_time,
    &this_update_time,
    &next_update_time);
if (res != 1) {
```

```
    if (error_stream)
        fprintf(
            error_stream,
            "Server certificate status is not found "
            " in the OCSP response\n");
    goto cleanup;
}
```

The `OCSP_resp_find_status()` call that we've just made provided us with information about the revocation status of the certificate, the time when the OCSP response was produced, and when the next OCSP response for the same certificate must be fetched.

9. Let's check whether the response is still valid:

```
res = OCSP_check_validity(
    this_update_time, next_update_time, 300, -1);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "OCSP response is outdated\n");
    goto cleanup;
}
```

10. Now that we've validated the OCSP response, let's check the certificate revocation status and change the `exit_code` variable if the certificate has been revoked:

```
switch (revocation_status) {
    case V_OCSP_CERTSTATUS_REVOKED:
        if (error_stream)
            fprintf(
                error_stream,
                "Server certificate is revoked\n");
        exit_code = 0;
        break;
    case V_OCSP_CERTSTATUS_GOOD:
        if (error_stream)
            fprintf(
                error_stream,
                "Server certificate is not revoked\n");
```

```
            break;
        default:
            if (error_stream)
                fprintf(
                    error_stream,
                    "Server certificate revocation status "
                    "is unknown\n");
    }
```

11. We will finish the OCSP stapling callback function by freeing the objects that are not needed anymore and returning the exit code:

```
    cleanup:
        if (server_cert_id)
            OCSP_CERTID_free(server_cert_id);
        if (ocsp_basicresp)
            OCSP_BASICRESP_free(ocsp_basicresp);
        if (ocsp_response)
            OCSP_RESPONSE_free(ocsp_response);
        return exit_code;
    }
```

The complete source code for our `ocsp-check` program can be found in this book's GitHub repository in the `ocsp-check.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/ocsp-check.c`.

Now that we've finished implementing the `ocsp-check` program, let's run it.

Running the program

We can run our `ocsp-check` program like so:

```
$ ./ocsp-check www.example.org 443
* verify_callback() called with depth=2, preverify_ok=1, error_
code=0, error_string=ok
  Certificate Subject: C = US, O = DigiCert Inc, OU = www.
digicert.com, CN = DigiCert Global Root CA
* verify_callback() called with depth=1, preverify_ok=1, error_
code=0, error_string=ok
  Certificate Subject: C = US, O = DigiCert Inc, CN = DigiCert
TLS RSA SHA256 2020 CA1
```

```
* verify_callback() called with depth=0, preverify_ok=1, error_
code=0, error_string=ok
  Certificate Subject: C = US, ST = California, L = Los
Angeles, O = Internet Corporation for Assigned Names and
Numbers, CN = www.example.org
* ocsp_callback() called with the following OCSP response:
  -----
  OCSP Response Data:
    …
    Cert Status: good
    …
  -----
  Server certificate is not revoked
TLS communication succeeded
```

As we can see, our `ocsp-check` program can successfully check the server certificate revocation status via OCSP.

In the previous sections, we learned how to extensively verify a TLS server certificate by using a custom verification callback, CRL, and OCSP. Those were important aspects of using X.509 certificates in the TLS protocol. In the next section, we will learn about another important aspect of using certificates in TLS: TLS client certificates.

Using TLS client certificates

In this section, we will learn how to use TLS client certificates: generate them, package them into **Public Key Cryptography Standards #12** (**PKCS #12**) containers, request them from TLS servers, and supply them to TLS clients.

Generating TLS client certificates

Apart from a TLS server, a TLS client can also provide a certificate on a TLS connection. However, according to the TLS protocol, the TLS client does not send its certificate by default, even if the client has it. The TLS client only sends its certificate if the TLS server requests it.

Another peculiarity of TLS client certificates is that a client certificate is often stored together with its certificate private key (or rather a key pair) in the PKCS #12 format. PKCS #12 is a file format that can store several cryptographic objects, such as X.509 certificates and key pairs. These stored objects can be encrypted using a symmetric cipher and authenticated using a **Hash-based Message Authentication Code** (**HMAC**). A typical PKCS #12 file with a TLS client certificate is protected by a password and contains the client certificate, its key pair, and the CA certificates that form the client certificate verification chain. Even though a typical PKCS #12 file contains more than a client certificate, it is usually called a client certificate file.

The filename extensions for PKCS #12 files are `.p12` and `.pfx`. If you want to use a TLS client certificate in a popular web browser, such as Mozilla Firefox or Google Chrome, you will need to supply the client certificate to the browser in the PKCS #12 container file, preferably with the `.p12` or `.pfx` extension. We are going to read a client certificate from a PKCS #12 file in an upcoming code example.

Let's learn how to generate a client certificate and package it into the PKCS #12 container.

We will generate a client certificate very similar to how we generated a server certificate in *Chapter 9*, *Establishing TLS Connections and Sending Data over Them*. We will also reuse the root CA certificate and its key pair from the previous chapter:

```
$ openssl req \
    -newkey ED448 \
    -subj "/CN=Client certificate" \
    -addext "basicConstraints=critical,CA:FALSE" \
    -noenc \
    -keyout client_keypair.pem \
    -out client_csr.pem
$ openssl x509 \
    -req \
    -in client_csr.pem \
    -copy_extensions copyall \
    -CA ca_cert.pem \
    -CAkey ca_keypair.pem \
    -days 3650 \
    -out client_cert.pem
```

After running the specified commands, we will have a client certificate and its key pair saved to the PEM files. In the next section, we will learn how to package them into a PKCS #12 container.

Packaging client certificates into PKCS #12 container files

To package the generated client certificate, its key pair, and the CA certificate into a PKCS #12 container, we can use the `openssl pkcs12` subcommand of the `openssl` command-line tool. The documentation for this subcommand is available on the `openssl-pkcs12` man page:

```
$ man openssl-pkcs12
```

Here's how we package our key pair and certificates into a PKCS #12 container:

```
$ openssl pkcs12 \
    -export \
```

```
    -inkey client_keypair.pem \
    -in client_cert.pem \
    -certfile ca_cert.pem \
    -passout 'pass:SuperPa$$w0rd' \
    -out client_cert.p12
```

Note that we supplied a password for the PKCS #12 container – that is, `SuperPa$$w0rd`. It is not very secure to supply a password on the command line because the command may be saved in the command's history and may be visible in the process list. In this example, we just did this for simplicity. Note that the `openssl` tool supports more secure ways to supply a password, such as reading a password from a file or stdin. You can read more about supplying a password to the `openssl` tool on the following `man` page:

```
$ man openssl-passphrase-options
```

With that, we have successfully created the PKCS #12 container file. Now, we can check what's inside:

```
$ openssl pkcs12 \
    -in client_cert.p12 \
    -passin 'pass:SuperPa$$w0rd' \
    -noenc \
    -info
MAC: sha256, Iteration 2048
MAC length: 32, salt length: 8
PKCS7 Encrypted data: PBES2, PBKDF2, AES-256-CBC, Iteration
2048, PRF hmacWithSHA256
Certificate bag
Bag Attributes
    localKeyID: 63 8B 00 96 4A 4D B7 E9 AF BD C2 09 A5 4A B8 3D
A2 FB 40 85
subject=CN = Client certificate
issuer=CN = Root CA
-----BEGIN CERTIFICATE-----
… base64-encoded data …
-----END CERTIFICATE-----
Certificate bag
Bag Attributes: <No Attributes>
subject=CN = Root CA
issuer=CN = Root CA
```

```
-----BEGIN CERTIFICATE-----
… base64-encoded data …
-----END CERTIFICATE-----
PKCS7 Data
Shrouded Keybag: PBES2, PBKDF2, AES-256-CBC, Iteration 2048,
PRF hmacWithSHA256
Bag Attributes
    localKeyID: 63 8B 00 96 4A 4D B7 E9 AF BD C2 09 A5 4A B8 3D
A2 FB 40 85
Key Attributes: <No Attributes>
-----BEGIN PRIVATE KEY-----
… base64-encoded data …
-----END PRIVATE KEY-----
```

As we can see, the produced PKCS #12 container file contains the client certificate, the CA certificate, and the client certificate key pair. The container's contents are encrypted with AES-256-CBC and authenticated with HMAC-SHA256.

Now that we have prepared the client certificate, let's implement a small TLS server program that will request and verify it.

Requesting and verifying a TLS client certificate on the server side programmatically

In this section, we are going to write a small TLS server program that requests and verifies a TLS client certificate. We are going to base it on the `tls-server` program from *Chapter 9*, *Establishing TLS Connections and Sending Data over Them*. We will take the code of the `tls-server` program and develop it further. Our new program will be called `tls-server2`.

Our `tls-server2` program will take the following command-line parameters:

1. Server port
2. The name of the file containing the TLS server key pair
3. The name of the file containing the TLS server certificate chain
4. The name of the file containing a trusted CA certificate that can verify the client certificate

To be able to verify a client certificate, we have to load the corresponding trusted CA certificate and enable peer certificate verification.

Verifying the TLS client certificate

Here is how we verify the client certificate:

1. First, we have to load the needed trusted CA certificate. We have to capture the fourth command-line argument and send it to the `run_tls_server()` function. Let's add the following line to the `main()` function:

```
const char* trusted_cert_fname = argv[4];
```

2. We must also pass the `trusted_cert_fname` parameter to the `run_tls_server()` function and add the following code inside the `run_tls_server()` function after the code that loads and checks the private key:

```
err = SSL_CTX_load_verify_locations(
    ctx, trusted_cert_fname, NULL);
if (err <= 0) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not load trusted certificates\n");
    goto failure;
}
```

3. We also have to change the following code to request the client certificate and enable its verification:

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
```

The `SSL_VERIFY_PEER` flag instructs OpenSSL to request the client certificate. Note that if the client did not supply a certificate, the absence of the certificate will not be considered an error on the server side. This behavior can be changed by setting the `SSL_VERIFY_FAIL_IF_NO_PEER_CERT` flag together with the `SSL_VERIFY_PEER` flag, like this:

```
SSL_CTX_set_verify(
    ctx,
    SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT,
    NULL);
```

If both the `SSL_VERIFY_PEER` and `SSL_VERIFY_FAIL_IF_NO_PEER_CERT` flags are set, and the client did not supply a certificate, then the TLS handshake will fail.

This code is enough to request the client certificate and verify it. However, it would also be nice to print the client certificate Subject, to make sure that the TLS client has sent its certificate and the TLS server has verified it. To do this, we will introduce a `construct_response()` function, which will construct a server response that will contain information about the client certificate. In `tls-server2`, we will use a response created by that function instead of the static response that we used in the original `tls-server` program.

Implementing the response generation function

The function will have the following signature:

```
BIO* construct_response(SSL* ssl);
```

The `ssl` parameter is needed to get the client certificate. The function will return a memory BIO that will contain the server response.

Let's implement the `construct_response()` function:

1. We will start by creating a memory BIO and printing the server response headers there:

```
BIO* mem_bio = BIO_new(BIO_s_mem());
const char* response_headers =
    "HTTP/1.0 200 OK\r\n"
    "Content-type: text/plain\r\n"
    "Connection: close\r\n"
    "Server: Example TLS server\r\n"
    "\r\n";
BIO_puts(mem_bio, response_headers);
```

2. The next step is to get the client certificate:

```
X509* peer_cert = SSL_get_peer_certificate(ssl);
```

   Note that in our example, we are not using the `SSL_VERIFY_FAIL_IF_NO_PEER_CERT` flag, which is why the TLS client can choose not to supply a certificate. In such a case, the `peer_cert` pointer will be `NULL`. We have to take this into account later.

3. The next step is to print information about the client certificate or its absence to the memory BIO:

```
if (peer_cert) {
    X509_NAME* peer_cert_subject =
        X509_get_subject_name(peer_cert);
    BIO_puts(
        mem_bio,
```

```
        "The TLS client certificate subject:\n");
    X509_NAME_print_ex(
        mem_bio,
        peer_cert_subject,
        0,
        XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB);
    BIO_puts(mem_bio, "\n");
    X509_free(peer_cert);
} else {
    BIO_puts(
        mem_bio,
        "The TLS client has not provided a
certificate\n");
}
```

4. We finish the `construct_response()` function by returning the memory BIO:

```
return mem_bio;
```

Very nice – we have implemented the `construct_response()` function! We will call that function from the `handle_accepted_connection()` function, instead of the code that constructed the static response and calculated its length.

5. The following code is from the original `tls-server` program:

```
const char* response =
    "HTTP/1.0 200 OK\r\n"
    "Content-type: text/plain\r\n"
    "Connection: close\r\n"
    "Server: Example TLS server\r\n"
    "\r\n"
    "Hello from the TLS server!\n";
int response_length = strlen(response);
```

We must replace it with the following code:

```
BIO* mem_bio = construct_response(ssl);
char* response = NULL;
long response_length = BIO_get_mem_data(mem_bio,
&response);
```

6. Don't forget to free the memory BIO that we have introduced in our new code! To free it, we must add the following code at the end of the `handle_accepted_connection()` function:

```
if (mem_bio)
    BIO_free(mem_bio);
```

Those were all the changes that we needed to make to the TLS server code.

The complete source code for our `tls-server2` program can be found in this book's GitHub repository in the `tls-server2.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/tls-server2.c`.

Running the program

Let's run our `tls-server2` program and see how it works:

1. We have not written a TLS client program that supplies a client certificate yet. However, we can use the `openssl s_client` subcommand or the `curl` utility for testing. We will need two terminal windows – one for the TLS server and another for the TLS client. Let's start our `tls-server2` program in the first terminal window:

```
$ ./tls-server2 \
    4433 \
    server_keypair.pem \
    server_cert.pem \
    ca_cert.pem
*** Listening on port 4433
```

2. Let's launch the `openssl` tool as a TLS client with a client certificate in the second terminal window:

```
$ openssl s_client \
    -connect localhost:4433 \
    -CAfile ca_cert.pem \
    -key client_keypair.pem \
    -cert client_cert.pem
```

3. When `openssl` is launched, it will connect to `tls-server2` and output a lot of information. Just press the *Enter* key after. The `openssl` tool will output the following:

```
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
```

```
Server: Example TLS server
The TLS client certificate subject:
CN = Client certificate
```

4. Let's examine the first terminal window where `tls-server2` is running. We will see the following output:

```
*** Receiving from the client:
*** Receiving from the client finished
*** Sending to the client:
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
The TLS client certificate subject:
CN = Client certificate
*** Sending to the client finished
```

As we can see, our `tls-server2` program has requested, received, and successfully verified the client certificate, as well as reported the client certificate's Subject.

5. We can perform the same test using the `curl` utility. Let's run it in the second terminal window:

```
$ curl \
    https://localhost:4433 \
    --cacert ca_cert.pem \
    --cert 'client_cert.p12:SuperPa$$w0rd' \
    --cert-type P12
The TLS client certificate subject:
CN = Client certificate
```

Note that we have supplied the client certificate to `curl` in PKCS #12 format.

As we can see, our `tls-server2` program can provide a nice report about the client certificate if the TLS client supplies it. But what will happen if the TLS client does not supply a certificate?

6. Let's check this using the `openssl` tool. Let's launch `openssl s_client` without the certificate:

```
$ openssl s_client \
    -connect localhost:4433 \
    -CAfile ca_cert.pem
```

7. After connecting and pressing *Enter*, we will get the following report:

```
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
The TLS client has not provided a certificate
```

8. We will get a similar report using `curl`:

```
$ curl https://localhost:4433 --cacert ca_cert.pem
The TLS client has not provided a certificate
```

From this, we can conclude that our `tls-server2` program works as expected. It can request, verify, and report information about the TLS client certificate, as well as gracefully handle the absence of the client certificate.

The `tls-server2` program will run and accept connections until it is aborted. You can abort it by pressing *Ctrl* + *C* in the terminal window where it is running.

With that, we have learned how to request a client certificate from the server side and verify it. In the next section, we will learn how to load a TLS client certificate onto the client side and use it for the TLS connection.

Establishing a TLS client connection with a client certificate programmatically

In this section, we are going to write a small TLS client program that uses a TLS client certificate. We are going to base it on the `tls-client` program from *Chapter 9*, *Establishing TLS Connections and Sending Data over Them*. Here, we will take the code of the `tls-client` program and develop it further. Our new program will be called `tls-client2`.

Our `tls-client2` program will take the following command-line parameters:

1. Server hostname
2. Server port
3. Name of the file containing the trusted CA certificate for server certificate verification
4. Name of the PKCS #12 file containing the client certificate and its key pair
5. Password for the PKCS #12 file, along with the client certificate

Compared to the original `tls-client` program, `tls-client2` takes two more parameters. All five parameters are now mandatory.

We are going to load the client certificate from a PKCS #12 container file. It is also possible to load the client certificate and its key pair from PEM files using the `SSL_CTX_use_certificate_chain_file()` and `SSL_CTX_use_PrivateKey_file()` functions, as we do in the `tls-server` and `tls-server2` programs. However, I want to demonstrate loading from a PKCS #12 file because client certificates are often distributed in PKCS #12 files.

We are going to use some OpenSSL functions that we have not used before. Here are the relevant `man` pages, along with the documentation for them:

```
$ man BIO_new_file
$ man d2i_PKCS12_bio
$ man PKCS12_parse
$ man SSL_CTX_use_cert_and_key
$ man PKCS12_free
```

Now, let's start implementing the `tls-client2` program with the changes to the code that we inherited from the `tls-client` program.

Changing the code inherited from the tls-client program

Here is what we need to change:

1. As we have two more command-line parameters, we should capture them in the named variables in the `main()` function:

   ```
   const char* client_cert_fname    = argv[4];
   const char* client_cert_password = argv[5];
   ```

2. We also have to pass the two new parameters to the `run_tls_client()` function. Hence, the `run_tls_client()` function will look like this:

   ```
   int run_tls_client(
       const char* hostname,
       const char* port,
       const char* trusted_cert_fname,
       const char* client_cert_fname,
       const char* client_cert_password,
       FILE* error_stream);
   ```

   The `run_tls_client()` call in the `main()` function will look like this:

   ```
   run_tls_client(
       hostname,
   ```

```
    port,
    trusted_cert_fname,
    client_cert_fname,
    client_cert_password,
    stderr);
```

3. Inside the `run_tls_client()` function, after loading the trusted certificates, we must add a call to the `load_client_certificate()` function that loads the client certificate into the `SSL_CTX` object:

```
int load_exit_code = load_client_certificate(
    client_cert_fname,
    client_cert_password,
    ctx,
    error_stream);
if (load_exit_code != 0)
    goto failure;
```

Now, we have to implement the `load_client_certificate` function.

Loading the TLS client certificate

The `load_client_certificate` function will have the following signature:

```
int load_client_certificate(
    const char* client_cert_fname,
    const char* client_cert_password,
    SSL_CTX* ctx,
    FILE* error_stream);
```

The function will return `0` on success and a non-zero exit code on failure.

Let's proceed with the function's implementation:

1. First, we must define the function exit code, which can be changed if a failure occurs:

```
int exit_code = 0;
```

2. Then, we must create a file BIO and open the client certificate file for reading:

```
BIO* client_cert_bio = BIO_new_file(client_cert_fname,
"rb");
if (!client_cert_bio) {
```

```
    if (error_stream)
        fprintf(
            error_stream,
            "Could not open client certificate file
%s\n",
            client_cert_fname);
    goto failure;
}
```

3. The next step is to load the client certificate file into the `PKCS12` object:

```
PKCS12* pkcs12 = d2i_PKCS12_bio(client_cert_bio, NULL);
if (!pkcs12) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not load client certificate "
            "from file %s\n",
            client_cert_fname);
    goto failure;
}
```

Note that we could open the client certificate file using `fopen()` and load it using `d2i_PKCS12_fp()`. However, I wanted to demonstrate the usage of yet another BIO type – the `file` BIO.

4. The `d2i_PKCS12_bio()` function has loaded the PKCS #12 container but did not decrypt or parse it. The next step is to verify the password so that we can decrypt the PKCS #12 container. This step is optional, but it can be useful for troubleshooting:

```
int res = PKCS12_verify_mac(
    pkcs12,
    client_cert_password,
    strlen(client_cert_password));
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "Invalid password was provided "
            "for client certificate file %s\n",
```

```
            client_cert_fname);
    goto failure;
}
```

5. After we have verified the PKCS #12 container password, we must try to decrypt it, and then parse and get the client certificate, its key pair, and CA certificates for the verification chain:

```
res = PKCS12_parse(
    pkcs12,
    client_cert_password,
    &pkey,
    &cert,
    &cert_chain);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not decode client certificate "
            "loaded from file %s\n",
            client_cert_fname);
    goto failure;
}
```

6. If the PKCS #12 container has been parsed successfully, we can set the client certificate, its key pair, and the CA certificates to the `SSL_CTX` object:

```
res = SSL_CTX_use_cert_and_key(
    ctx,
    cert,
    pkey,
    cert_chain,
    1);
if (res != 1) {
    if (error_stream)
        fprintf(
            error_stream,
            "Could not use client certificate "
            "loaded from file %s\n",
```

```
            client_cert_fname);
    goto failure;
}
```

Note that the `SSL_CTX_use_cert_and_key()` function will increase the reference count on the `cert`, `pkey`, and `cert_chain` objects. Thus, we will have to free them at the end of the function. As the `SSL_CTX` object will still hold references for them, the mentioned objects will not be deallocated on that freeing at the end of the function. Instead, they will be deallocated when the `SSL_CTX` object itself is deallocated.

7. When it comes to potential troubleshooting, it's a good idea to check that the loaded client certificate key pair matches the client certificate:

```
res = SSL_CTX_check_private_key(ctx);
if (res != 1) {
    if (error_stream)
        fprintf(
        error_stream,
        "Client keypair does not match "
        "client certificate\n");
    goto failure;
}
```

At this point, we have successfully loaded the client certificate, its key pair, and the CA certificate, and set it to the `SSL_CTX` object.

8. Now, let's finish up the `load_client_certificate()` function code by freeing the used objects and returning the exit code:

```
    goto cleanup;
failure:
    exit_code = 1;
cleanup:
    if (cert_chain)
        sk_X509_pop_free(cert_chain, X509_free);
    if (cert)
        X509_free(cert);
    if (pkey)
        EVP_PKEY_free(pkey);
    if (pkcs12)
        PKCS12_free(pkcs12);
```

```
    if (client_cert_bio)
        BIO_free(client_cert_bio);
    return exit_code;
}
```

Note that the `BIO_free(client_cert_bio)` call will close the underlying file before deallocating the file BIO.

With that, we have finished implementing the `load_client_certificate()` function and the whole `tls-client2` program. The complete source code for our `tls-client2` program can be found in this book's GitHub repository in the `tls-client2.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter10/tls-client2.c`.

Running the program

Now, let's try to run the `tls-client2` program and check how it works:

1. As in the previous example, we will need two terminal windows. In the first window, we will run the `tls-server2` program:

```
$ ./tls-server2 \
    4433 \
    server_keypair.pem \
    server_cert.pem \
    ca_cert.pem
*** Listening on port 4433
```

2. In the second terminal window, we will run our `tls-client2` program:

```
$ ./run-tls-client2.sh
*** Sending to the server:
GET / HTTP/1.1
Host: localhost
Connection: close
User-Agent: Example TLS client
*** Sending to the server finished
*** Receiving from the server:
HTTP/1.0 200 OK
Content-type: text/plain
Connection: close
Server: Example TLS server
```

```
The TLS client certificate subject:
CN = Client certificate
*** Receiving from the server finished
TLS communication succeeded
```

As we can see, our `tls-client2` could load the client certificate and use it in the TLS connection. The `tls-server2` program has reported that it has got the client certificate on the server side. From this, we can conclude that the `tls-client2` program works as expected.

With that, we have learned how to use TLS client certificates. Now, let's summarize this chapter.

Summary

This chapter was different from the previous ones. In the previous chapters, we covered one big topic per chapter. In this chapter, however, we covered several small topics and wrote more example programs.

In this chapter, we learned about how to verify the peer certificate during a TLS handshake. We also learned about the CRL and OCSP methods. Then, we learned about TLS client certificates and PKCS #12 containers. After that, we learned how to request and verify a TLS client certificate on the server side and how to load and use client certificates on the client side. We finished this chapter by connecting our TLS server and TLS client programs, both of which support TLS client certificates, to each other.

The knowledge you've gained from this chapter will help you perform advanced client and server certificate processing on TLS connections in your TLS-enabled programs.

In the next chapter, we will learn about more advanced and special uses of TLS.